|  |
| --- |
| **General Questions:Linux OS** |
| ***1. Can a thread acquire more than one lock (Mutex)?*** |
| Yes, it is possible that a thread is in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock. |
| ***2. Can a mutex be locked more than once?*** |
| A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a recursive mutex can be locked more than once (POSIX complaint systems), in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked. |
| ***3. What happens if a non-recursive mutex is locked more than once.*** |
| Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock. It is because no other thread can unlock the mutex. An operating system implementer can exercise care in identifying the owner of mutex and return if it is already locked by same thread to prevent deadlocks. |
| ***4. Are binary semaphore and mutex same?*** |
| No. We suggest to treat them separately, as it is explained signalling vs locking mechanisms. But a binary semaphore may experience the same critical issues (e.g. priority inversion) associated with mutex. We will cover these in later article. |
| A programmer can prefer mutex rather than creating a semaphore with count 1. |
| ***5. What is a mutex and critical section?*** |
| Some operating systems use the same word critical section in the API. Usually a mutex is costly operation due to protection protocols associated with it. At last, the objective of mutex is atomic access. There are other ways to achieve atomic access like disabling interrupts which can be much faster but ruins responsiveness. The alternate API makes use of disabling interrupts. |
| ***6. What are events?*** |
| The semantics of mutex, semaphore, event, critical section, etc… are same. All are synchronization primitives. Based on their cost in using them they are different. We should consult the OS documentation for exact details. |
| ***7. Can we acquire mutex/semaphore in an Interrupt Service Routine?*** |
| An ISR will run asynchronously in the context of current running thread. It is not recommended to query (blocking call) the availability of synchronization primitives in an ISR. The ISR are meant be short, the call to mutex/semaphore may block the current running thread. However, an ISR can signal a semaphore or unlock a mutex. |
| ***8. What we mean by “thread blocking on mutex/semaphore” when they are not available?*** |
| Every synchronization primitive has a waiting list associated with it. When the resource is not available, the requesting thread will be moved from the running list of processor to the waiting list of the synchronization primitive. When the resource is available, the higher priority thread on the waiting list gets the resource (more precisely, it depends on the scheduling policies). |
| ***9. Is it necessary that a thread must block always when resource is not available?*** |
| Not necessary. If the design is sure ‘what has to be done when resource is not available‘, the thread can take up that work (a different code branch). To support application requirements the OS provides non-blocking API. |
| For example POSIX pthread\_mutex\_trylock() API. When mutex is not available the function returns immediately whereas the API pthread\_mutex\_lock() blocks the thread till resource is available. |
|  |
|  |
|  |
| **Linux Platform device driver: ww.codeproject.com** |
|  |
| **Introduction** |
| A platform device is one which is hardwired on the board and hence not hot-pluggable. The driver for this device need not check for the presence of the device and can just go on and do what is required to enable the device to make it operational. If the device is not found, the driver is simply ignored. |
| **Background** |
| Platform device driver works a bit differently than normal hot-pluggable device driver. Platform device is either directly mounted on the board or part of the SoC and hence can't be physically removed. Compared to hot-pluggable devices, platform device might require some extra settings like gpios pin muxing, power on & reset using gpio control, initializing the clock source, etc., which are very specific to the board or SoC - hence the name platform device driver. This tip gives a brief overview of how to write a platform device driver for Linux. |
| **Using the Code** |
| The source code for this article is given for reference. The platform device code is usually part of the BSP/CSP. The device driver code is written as a separate module in the appropriate linux-kernel-src/driver/ directory (char for serial device, net for networking device, etc). In the given example code, both the platform device and platform driver code are presented in a single file for easy viewing of the code. |
| **Platform Data** |
| This structure is specific to the platform device. As indicated in the previous section, since platform device is usually part of the BSP/CSP, this data structure apart from class specific device data, will contain platform specific data such as gpios used, irq number (if interrupt is used), clock frequency settings required by the platform device etc. It usually implements functions that can be called by the device driver for doing platform specific stuff. This way, the device driver can be made independent of the platform and hence can be used across multiple platforms. For example, the below example implements functions for power ON/OFF the device and for doing the device reset. |
| Hide   Copy Code/\* Data structure for the platform data of "my device" \*/ |
| struct my\_device\_platform\_data { |
| int reset\_gpio; |
| int power\_on\_gpio; |
| void (\*power\_on)(struct my\_device\_platform\_data\* ppdata); |
| void (\*power\_off)(struct my\_device\_platform\_data\* ppdata); |
| void (\*reset)(struct my\_device\_platform\_data\* pdata); |
| }; |
| /\* "my device" platform data \*/ |
| static struct my\_device\_platform\_data my\_device\_pdata = { |
| .reset\_gpio = 100, |
| .power\_on\_gpio = 101, |
| .power\_on = my\_device\_power\_on, |
| .power\_off = my\_device\_power\_off, |
| .reset = my\_device\_reset |
| }; |
| **Registering Platform Device** |
| This is done by calling platform\_device\_register() function with the device instance as shown below. The name of the device is very important as this string is used by the OS for calling the probe() function when the corresponding driver is installed. Please note that this function must be called before registering Platform Driver. |
| Hide   Copy Codestatic struct platform\_device my\_device = { |
| .name = "my-platform-device", |
| .id = PLATFORM\_DEVID\_NONE, |
| .dev.platform\_data = &my\_device\_pdata |
| }; |
| platform\_device\_register(&my\_device); |
| **Platform Driver** |
| The platform driver implements a probe function that is called by the OS when this driver is inserted. Since the driver is supposed to be platform independent, it depends on the platform\_data functions to set up the device and make it operational. |
| Hide   Copy Codestatic struct platform\_driver my\_driver = { |
| .probe = my\_driver\_probe, |
| .remove = my\_driver\_remove, |
| .driver = { |
| .name = "my-platform-device", |
| .owner = THIS\_MODULE, |
| .pm = &my\_device\_pm\_ops, |
| }, |
| }; |
| **Registering Platform Driver** |
| Here is the function for registering driver with the OS. Please note that we are using platform\_driver\_probe() instead of platform\_driver\_register(), since we know that this device is present for sure in the system. The difference between these two functions is that with platform\_driver\_register(), we are asking the OS to put this driver in the list of drivers it maintains for doing device to driver matching when the devices come in/out of the system. Since platform devices are either always present always absent in system (and not-hot-pluggable), we don't need to put our platform driver in the OS driver list. With platform\_driver\_probe(), we are asking the OS to check if a platform device is present with the matching name. If the device is present in the system, the corresponding probe() function is called. If not present, the driver is simply ignored. |
| Hide   Copy Coderet = platform\_driver\_probe(&my\_driver, my\_driver\_probe); |
| **Platform Device and Driver binding** |
| When the platform\_driver\_probe() function is called, the operating system scans through the list of available platform devices and checks if the driver.name matches with the device name. If yes, it calls the driver probe function with the platform data. The probe function can then initialize the clock settings, power-on, reset the device, allocated the driver data and register the corresponding class specific driver with the OS. |
| Hide   Shrink   Copy Codestatic int my\_driver\_probe(struct platform\_device \*pdev) |
| { |
| struct my\_device\_platform\_data \*my\_device\_pdata; |
| struct my\_driver\_data\* driver\_data; |
| printk(KERN\_ALERT " %s\n", \_\_FUNCTION\_\_); |
| my\_device\_pdata = dev\_get\_platdata(&pdev->dev); |
| /\* Power on the device. \*/ |
| if (my\_device\_pdata->power\_on) { |
| my\_device\_pdata->power\_on(my\_device\_pdata); |
| } |
| /\* wait for some time before we do the reset \*/ |
| mdelay(5); |
| /\* Reset the device. \*/ |
| if (my\_device\_pdata->reset) { |
| my\_device\_pdata->reset(my\_device\_pdata); |
| } |
| /\* Create the driver data here \*/ |
| driver\_data = kzalloc(sizeof(struct my\_driver\_data), GFP\_KERNEL); |
| if(!driver\_data) |
| return -ENOMEM; |
| /\* Set this driver data in platform device structure \*/ |
| platform\_set\_drvdata(pdev, driver\_data); |
|  |
| /\* Call the class specific register driver here \*/ |
| // tty\_register\_driver(ttyprintk\_driver); // for UART driver |
| // register\_netdev(net\_driver); // for Net driver |
| // sdio\_register\_driver(sdio\_river) |
| return 0; |
| } |